UNIVERSITÀ DEGLI STUDI DI PADOVA

Facoltà di Ingegneria

Corso di Laurea Triennale in Ingegneria Informatica

Tesi di Laurea

UNA FORMULAZIONE MIP PER ROADEF/EURO 2011-2012 (MACHINE REASSIGNMENT)

Relatore

Prof. Domenico Salvagnin

Laureando

Veronica Polonio

26 luglio 2012

Indice

Capitolo 1

Introduzione

La *ricerca operativa* è una disciplina che ha come oggetto lo studio di metodologie per la risoluzione di problemi decisionali. I sistemi studiati possono essere modellati in modo matematico a partire da una descrizione del problema. L'obbiettivo è arrivare ad una soluzione che puó essere ottima, quando è possibile calcolarla, oppure approssimata. La ricerca operativa ha un ruolo importante nelle attivitá decisionali, raggiunge un determinato obiettivo rispettando vincoli che sono imposti dall'esterno e non sono sotto il controllo di chi deve compiere le decisioni. Una delle branche della ricerca operativa è l'*ottimizzazione*. Essa si occupa di massimizzare o minimizzare una funzione obbiettivo sottoposta a dei vincoli.

In questo elaborato si è voluto applicare i principi della programmazione lineare intera per risolvere un problema di ottimizzazione assegnato da *Roadef*, una societá francese che organizza ogni anno una competizione su temi di interesse di ricerca operativa.

Il testo è stato strutturato in tre capitoli. Nella prima parte viene descritto la struttura del problema e le specifiche che deve rispettare. Nella seconda è stato realizzato il modello algebrico. Nell'ultima sezione sono stati analizzati i risultati ottenuti. Infine nell'appendice è possibile trovare un esempio dei parametri forniti, i sorgenti utilizzati per la risoluzione del modello e i sorgenti con cui sono stati elaborati i parametri iniziali.

Tutte le istanze numeriche utilizzate ed il testo integrale del problema sono reperibili all'indirizzo *http://challenge.roadef.org/2012*.

1.1 Programmazione lineare intera

In molti problemi di ottimizzazione, le variabili decisionali hanno senso solo se assumono valori interi. Un problema di *programmazione lineare intera (MIP)* consiste nella minimizzazione di una funzione lineare soggetta ad un numero finito di vincoli lineari, con in piú il vincolo che alcune variabili devono assumere valori interi. La forma generica di un problema MIP è:

$$\begin{aligned} min \quad & cx \\ & Ax \leq b \\ & x \in \mathbb{Z}^n \end{aligned} \tag{1.1}$$

Ad esempio, spesso è necessario assegnare persone, macchine o veicoli a varie attivitá in quantitá intere. Quando si pretende che alcune variabili assumano valori interi si tratta di problemi di programmazione lineare intera. Molti problemi MIP richiedono, al caso pessimo, un tempo di computazione della soluzione esponenziale rispetto alla dimensione dei dati in ingresso. In pratica si tratta di problemi difficili da risolvere. Tuttavia alcuni di questi problemi, di uso comune nelle applicazioni, sono stati studiati a fondo e oggi esistono delle tecniche che permettono di risolverli in tempo ragionevole nelle applicazioni piú comuni.

Quando in un problema MIP si eliminano i vincoli $x \in \mathbb{Z}^n$ si ottiene un problema di programmazione lineare detto rilassamento continuo.

1.2 Algoritmo Branch and Bound

Il *branch-and-bound* (B&B) è una tecnica generale per la risoluzione di problemi di ottimizzazione. Si basa su una enumerazione intelligente dell'insieme F delle soluzioni ammissibili di un problema di ottimizzazione combinatoria. L'algoritmo B&B ottimizza la ricerca sfruttando un principio molto semplice: non esplorare una regione dello spazio delle soluzioni se si puó dimostrare che non contiene soluzioni migliori di quelle note.

Il concetto alla base dell'algoritmo B&B è *divide and conquer*. Essendo difficile risolvere direttamente il problema originale esso viene diviso in problemi piú piccoli, che poi vengono ottimizzati separatamente.

Definiamo F come l'insieme delle soluzioni ammissibili di un problema di minimizzazione, $c : F \to \mathbb{R}$ la funzione obbiettivo da minimizzare e $\bar{x} \in F$ una soluzione ammissibile nota, chiamata *incumbent*. Il costo $z = c(\bar{x})$ rappresenta un upper-bound sul costo della soluzione ottima.

La prima fase dell'algoritmo B&B consiste nella rilassare il problema ammettendo soluzioni che non appartengono all'insieme F, ma ad un suo sovrainsieme $G \supseteq F$. La soluzione del rilassamento fornisce dunque un lower bound sul valore della soluzione ottima. Se la soluzione del rilassamento appartiene a F o ha un costo uguale a z si ha finito : la nuova soluzione (se ammissibile) o quella nota è ottima.

Altrimenti si identifica una separazione F^* di F, cioé un insieme finito F^* di sottoinsiemi di F, tale che:

$$\bigcup_{F_i \in F^*} F_i = F$$

Ogni sottoinsieme F_i è figlio di F. Questa fase, detta di *branching*, è giustificata dal fatto che se F^* è una separazione, allora :

$$min\{c(x) \,|\, x \in F\} = min\{min\{c(x) \,|\, x \in F_i\} F_i \in F^*\}$$

F^* è spesso, ma non necessariamente, una partizione di F. I figli di F vengono aggiunti alla coda A dei sottoproblemi da processare.

Successivamente si seleziona un sottoproblema P_i dalla coda e se ne risolve un rilassamento. Si possono verificare quattro casi:

- Se si dimostra che lo soluzione del sottoproblema è ottima, si procede con la risoluzione degli altri sottoproblemi.
- Se un sottoproblema è impossibile si scarta (*prunning by infeasibility*)
- Se il valore del rilassamento è maggiore o uguale all'upper bound globale z il sottoproblema si puó scartare (*prunning by optimality*).
- Se non è possibile scartare il problema lo si suddivide ulteriormente e si aggiungono i figli del sottoproblema alla lista dei sottoproblemi da processare.

Si continua in questo modo finché la lista dei sottoproblemi è vuota.

La scelta del rilassamento è una scelta molto importante per un algoritmo branch-and-bound. Deve essere scelto in modo da essere relativamente facile da risolvere e dare allo stesso tempo un lower bound ragionevole.

L'algoritmo è corretto anche senza un incumbent, anche se in genere è molto meno efficace perché è possibile scartare un sottoproblema solo per feasibility e non per optimality.

Quando si utilizza l'algoritmo B&B appenda descritto, si deve specificare come calcolare i rilassamenti e le separazioni. La scelta piú diffusa consiste

nel calcolare il rilassamento lineare. Se la soluzione di tale rilassamento non è intera possiamo scegliere una x_j con valore frazionario x_j^* nella soluzione e considerare la seguente partizione :

$$x_j \leq \lfloor x_j^* \rfloor \vee x_j \geq \lceil x_j^* \rceil$$

1.3 Algoritmo Branch and Cut

Ad ogni nodo dell'albero decisionale, la formulazione associata ad ogni rilassamento del sottoproblema corrente puó essere rafforzata mediante la generazione di *piani di taglio*. Un piano di taglio è un vincolo che riduce la regione ammissibile per il rilassamento lineare senza eliminare delle soluzioni ammissibili. Tale rafforzamento è fatto per:

- ottenere una soluzione intera del rilassamento lineare
- ottenere un lower bound migliore e quindi piú efficace per il pruning

Per la risoluzione del modello di programmazione lineare intera si puó utilizzare l'algoritmo *branch-and-cut*. Questo algoritmo è stato ideato da Padberg e Rinaldi nel 1987.

Questa tecnica è spesso molto efficace e risolve molti difetti del B&B e a piani di taglio puri.

L'approccio B&C richiede l'esistenza di procedure efficienti per la risoluzione del seguente problema di *separazione* : data una soluzione frazionaria x^*, trovare una diseguaglianza valida $a^T x \leq \alpha_0$, se esiste, violata da x^*, cioé tale che $a^T x^* > \alpha_0$.

1.4 ILOG CPLEX Optimization Studio

ILOG è una societá francese, di proprietá di *IBM*, ed è una dei leader nel settore della Ricerca Operativa. Essa fornisce tra le migliori tecnologie di ottimizzazione a livello mondiale.

ILOG CPLEX Optimization Studio è un insieme di strumenti utilizzati per la modellazione e lo sviluppo di applicazioni di ottimizzazione. ILOG CPLEX Optimization Studio mette a disposizione:

- un solver per la risoluzione di problemi di programmazione lineare
- un solver per la risoluzione di problemi di programmazione con vincoli

- un solver per la risoluzione di problemi di programmazione quadratica convessa
- un linguaggio di modellazione algebrico: *OPL*

OPL (*Optimization Programming Language*) è un linguaggio di programmazione ad alto livello per l'ottimizzazione. È utilizzato per risolvere problemi di programmazione lineare, intera e mista e per i problemi di programmazione con vincoli.

Un programma scritto in OPL è generalmente strutturato in tre parti:

- una sequenza di dichiarazioni di costanti e di variabili;
- un'istruzione che indica la funzione obiettivo;
- una sequenza di dichiarazioni di vincoli;

Oltre al codice per la risoluzione del modello si possono inserire delle righe di codice prima e dopo la definizione del problema che prendono il nome rispettivamente di *preprocessing* e *postprocessing*. Queste funzioni aggiuntive che si possono inserire servono, ad esempio, per visualizzare il valore che assumono le variabili dopo l'esecuzione del modello.

Sebbene sia consentito scrivere in un solo file sia il modello sia i dati, OPL permette di tenere separati questi due termini, costruendo:

- un *file di modello*, di estensione *.mod*, che descrive la struttura logica del modello (indici, insiemi, variabili, funzione obbiettivo e vincoli)
- un *file di dati*, di estensione *.dat*, che contiene i valori numerici del problema.

Mantenendo separato il modella dai dati è possibile applicare lo stesso modello a dati diversi, senza cambiare il modello evitando cosí il rischio di introdurre errori.

OPL mette a disposizione un vocabolario ridotto, la sintassi è facilmente apprendibile e quindi il codice è di immediata comprensione per l'utente, il modello risulta quindi compatto e autodocumentato.

A differenza di altri linguaggi di modellazione algebrica OPL supporta solo i solver forniti da IBM.

Capitolo 2

Descrizione del problema

Lo scopo del progetto consiste nel migliorare l'assegnamento dei processi in varie macchine. Una macchina possiede varie risorse, ad esempio RAM e CPU, e i processi in esecuzione consumano queste risorse. Inizialmente ogni processo è assegnato ad una macchina ma per migliorarne l'uso i processi possono essere spostati. I possibili movimenti hanno un costo e sono limitati da dei vincoli. La soluzione del problema consiste in un nuovo assegnamento processo-macchina che soddisfi tutti i vincoli e minimizzi i costi.

Definiamo M come un insieme di macchine e P un insieme di processi. Una soluzione consiste nell'assegnare ogni processo $p \in P$ ad una e una sola macchina $m \in M$. L'assegnamento iniziale dei processi è determinato e realizzabile, infatti tutti i vincoli fissi sono rispettati.

2.1 Vincoli del problema

2.1.1 Vincoli di capacitá

Il vincolo di capacitá impone che un processo possa essere eseguito su una macchina se e solo se la macchina ha abbastanza capacitá disponibile per ogni risorsa.

Sia R l'insieme delle risorse che sono in comune a tutte le macchine , $C_{m,r}$ la capacitá della risorsa $r \in R$ per la macchina $m \in M$ e $R_{p,r}$ la richiesta della risorsa $r \in R$ per il processo $p \in P$. Dato un assegnamento l'uso U della macchina m per la risorsa r è definito come:

$$U_{m,r} = \sum_{p \in P \,:\, M_p = m} R_{p,r}$$

Il vincolo di capacitá impone:

$$\forall m \in M, \forall r \in R \quad U_{m,r} \leq C_{m,r}$$

2.1.2 Vincoli di conflitto

Il vincolo di conflitto richiede che i processi di un insieme, chiamato *servizio*, vengano eseguiti su macchine diverse.

Sia S un insieme di servizi e $s \in S$ un servizio. Tutti i servizi sono insiemi disgiunti. Formalmente il vincolo di conflitto puó essere definito come:

$$\forall s \in S,\ (p_i, p_j) \in s^2,\ p_i \neq p_j \Rightarrow M_{p_i} \neq M_{p_j}$$

2.1.3 Vincoli di spread

Una location è composta da un insieme di macchine. Il vincolo di spread esige che i processi appartenenti ad un servizio siano assegnati in $spreadMin_s \in \mathbb{N}$ location distinte.

Sia L l'insieme delle location e $l \in L$ una location. Le location sono insiemi disgiunti. Piú formalmente ii vincolo di spread puó essere espresso come:

$$\forall s \in S \quad \sum_{l \in L} min(1, \left|\{p \in s | M_p \in l\}\right|) \geq spreadMin_s$$

2.1.4 Vincoli di dipendenza

Definiamo un neightborhood come un insieme di macchine. Il vincolo di dipendenza impone che se un servizio s^a dipende dal servizio s^b, ogni processo del servizio s^a deve essere assegnato alle macchine appartenenti al neightborhood del servizio s^b.

Sia N un insieme di neightborhood e $n \in N$ un neightborhood. I neightborhood sono insiemi disgiunti.

Il vincolo di dipendenza puó essere espresso come:

$$p^a \in s^a, \exists\, p^b \in s^b \, e \, n \in \mathbb{N} \,:\, M_{p^a} \in n \, e \, M_{p^b} \in n$$

I vincoli di dipendenza non sono simmetrici, ad esempio se il servizio s^a dipende dal servizio s^b non è equivalente a dire che s^b dipenda da s^a

2.1.5 Vincoli di uso transitori

Quando un processo p è spostato da una macchina m a una macchina m' alcune risorse sono consumate due volte; per esempio lo spazio su disco

non è disponibile sulla macchina m durante la copia sulla macchina m', e m' dovrebbe avere abbastanza spazio su disco durante la copia. Sia $TR \subset R$ il sottoinsieme delle risorse che hanno bisogno le macchine durante lo spostamento dei processi. I vincoli d'uso transitorio sono dunque:

$$\forall m \in M, \forall r \in TR \quad \sum_{p \in P : M0_p = m \vee M_p = m} R_{p,r} \leq C_{m,r}$$

2.2 Funzione obiettivo

Lo scopo è di assegnare i processi alle macchine minimizzando i costi. Il costo finale è dato dalla combinazione di: load cost, balance cost,process move cost, service move cost e machine move cost.

Il costo totale è la somma di tutti i costi previsti.

$$\begin{aligned} totalcost &= \sum_{r \in R} weightLoadCost_r * loadCost_r \\ &+ \sum_{b \in B} weightBalanceCost_b * balanceCost_b \\ &+ weightProcessMoveCost * processMoveCost \\ &+ weightServiceMoveCost * serviceMoveCost \\ &+ weightMachineMoveCost * machineMoveCost \end{aligned}$$

2.2.1 Load cost

Il load cost è definito per ogni risorsa e corrisponde alla capacitá usata sopra la capacitá di sicurezza.

Sia $SC_{m,r}$ la capacitá di sicurezza della risorsa $r \in R$ nella macchina $m \in M$.

$$loadCost_r = \sum_{m \in M} max(0, U_{m,r} - SC_{m,r})$$

2.2.2 Balance cost

Ad esempio avendo a disposizione la risorsa CPU senza avere la risorsa RAM è inutile un futuro assegnamento di un processo in questa macchina. Un ulteriore obbiettivo è dunque bilanciare le risorse disponibili.

Sia B un insieme di triple definite in NxR^2.

Per ogni tripla $b =< r_1, r_2, target > \in B$ il balance cost è:

$$balanceCost_b = \sum_{m \in M} max(0, targetA_{m,r1} - A_{m,r2})$$

$$con \qquad A_{m,r} = C_{m,r} - U_{m,r}$$

2.2.3 Process move cost

Alcuni processi sono difficili da spostare; per tener conto di questo problema è stato introdotto il process move cost. Sia PMC_p il costo per lo spostamento del processo p dalla sua macchina iniziale $M0_p$, il process move cost risulta:

$$processMoveCost = \sum_{p \in P \,:\, M0_p \neq M_p} PMC_p$$

2.2.4 Service move cost

Per bilanciare lo spostamento dei servizi, viene definito il *service move cost* in cui viene assegnato il numero massimo di processi spostati all'interno di un servizio. Piú formalmente:

$$serviceMoveCost = max_{s \in S}(\Big|\{p \in s | M_p \neq M0_p\}\Big|)$$

2.2.5 Machine move cost

Il machine move cost esprime il costo dovuto dallo spostamento dei processi dalle rispettive macchine iniziali.

Sia $MMC_{msource,mdestination}$ il costo dello spostamento di ogni processo p dalla macchina m_{source} alla macchina $m_{destination}$. Ovviamente per ogni macchina $m \in M$ $MMC_{m,m} = 0$. Il machine move cost è quindi la somma di tutti gli spostamenti ponderati della relativa MMC.

$$machineMoveCost = \sum_{p \in P} MMC_{M0_p, M_p}$$

Capitolo 3

Modellazione algebrica

3.1 Introduzione alla modellazione algebrica

Il modello algebrico viene utilizzato per descrivere le caratteristiche della soluzione ottima di un problema di ottimizzazione attraverso relazioni matematiche. Oltre a fornire una descrizione formale del problema, il modello fornisce la base per l'applicazione di algoritmi di ottimizzazione in grado di determinare una soluzione ottima.

In generale le prime azioni da svolgere per la modellazione di un problema di ottimizzazione consistono nel determinare tutte le grandezze (insiemi, variabili, parametri) necessarie per descrivere il problema, di queste bisogna determinare quelle in cui i valori sono indipendenti dalla soluzione del problema (i dati) e quelle i cui valori dipendono proprio dalla soluzione del problema (le variabili).

I dati e le variabili non sono mai grandezze indipendenti, ma sono sempre legate da vincoli lineari che dipendono dalla struttura stessa del problema. Bisogna quindi identificare con precisione, nel modello, questi vincoli.

Infine si deve individuare l'obbiettivo che si vuole raggiungere. Quest'ultimo rappresenta la funzione obbiettivo che si vuole minimizzare o massimizzare. La funzione obbiettivo è un'espressione lineare delle variabili decisionali.

3.2 Insiemi

Gli insiemi definiscono il dominio del problema e la sua dimensione. Servono a raccogliere in una struttura dati (ad esempio un vettore) tutti i dati o le variabili che si riferiscono a termini omogenei.

Dal testo del problema si possono individuare 13 insiemi. Nella Tabella 3.1 vengono descritti.

Insiemi	**Definizione**
$M = \{m_1, ..., m_n\}$	Insieme delle macchine
$P = \{p_1, ...p_b\}$	Insieme dei processi
$R = \{r_1, ...r_c\}$	Insieme delle risorse
$S = \{s_1, ...s^a\}$	Insieme dei servizi
$L = \{l_1, ...l_i\}$	Insieme delle location
$N = \{n_1, ...n_k\}$	Insieme dei neightborhood
$TR \subset R$	Sottoinsieme di risorse
$B = \{b_1, ...b_h\}$	Insieme di triple per gli oggetti balance cost
$D = \{d_1, ...d_j\}$	Insieme di coppie per i vincoli di dipendenza
$s^i = \{p_1, ...p_r\}$	Insieme di processi per il servizio i
$l^i = \{m_1, ...m_s\}$	Insieme delle macchina per la location i
$n^i = \{m_1, ...m_z\}$	Insieme delle macchina per il neightborhood i

Tabella 3.1: Definizioni insiemi

3.3 Parametri

I parametri sono valori numerici che definiscono in dettaglio il problema che si vuole affrontare. Essi sono assegnati una volta per tutte.

Il problema fornisce dei dati che devono essere utilizzati per la risoluzione del modello. I parametri sono sintetizzati nella Tabella 3.2 .

Parametri	**Definizione**
$M0_p$	$M0_p = m$ Assegnamento iniziale del processo p alla macchina m
$M0B_{p,m} \in \{0, 1\}$	$M0B_p = 1$ Se l'assegnamento iniziale prevede che il processo assegnato p sia alla macchina m
$C_{m,r}$	Capacitá della risorsa r nella macchina m
$R_{p,r}$	Richiesta della risorsa r per il processo p
$spreadMin_s$	spreadMin relativo al servizio s
$SC_{m,r}$	Capacitá di sicurezza della risorsa r nella macchina m
PMC_p	Process move cost per il processo p
MMC_{MO_p,M_p}	Machine move cost del processo p dalla macchina iniziale alla macchina finale
$weightLoadCost_r$	Weight load cost per la risorsa r
$weightBalanceCost_p$	Weight balance cost per il processo p
$weightProcessMoveCost$	Weight process move cost
$weightServiceMoveCost$	Weight service move cost
$weightMachineMoveCost$	Weight machine move cost
$w_{p,m} \in \{0, 1\}$	$w_p = 1$ Se nell'assegnamento iniziale non prevede che il processo p sia assegnato alla macchina m
$b < r_1, r_2, target >$	Oggetti balance cost usati per bilanciare le risorse disponibili
$d < s^a, s^b >$	Dipendenza dal servizio s^a al servizio s^b

Tabella 3.2: Definizioni parametri

3.4 Variabili

Le variabili sono le grandezze che descrivono la soluzione del problema. Il loro valore deve essere determinato dal risolutore.

Sono state individuate classi di 14 variabili utili per la risoluzione del problema e sono sintetizzate nella Tabella 3.3. Si possono suddividere le variabili in due categorie: le variabili strutturali del problema e variabili necessarie per effettuare manipolazioni algebriche sul modello. Le variabili strutturali del modello sono le variabili $x_{p,m}$ utilizzate per determinare in quali macchine assegnare i processi. Tutte le altre variabili sono artificiali e sono state introdotte per modellare i vincoli del problema. In particolare :

- le variabili $bc_{m,b}$ e $balanceCost_b$ e sono state utilizzate per modellare il vincolo sul balance cost
- la variabile $U_{m,r}$ è stata utilizzata per modellare i vincoli di balance cost, load cost e capicitá
- le variabili $LC_{m,r}$ e $loadCost_r$ sono state utilizzate per modellare il vincolo sul load cost
- la variabile z_p è stata utilizzata per modellare i vincoli di process move cost e service move cost
- le variabili $serviceMoveCost$ e $processServiceMove_s$ sono state utilizzate per modellare il vincolo di service move cost
- la variabile $k_{s,l}$ è stata utilizzata per modellare il vincolo di spread
- la variabile $j_{p,n}$ è stata utilizzata per modellare il vincolo di dipendenza
- la variabile $y_{p,m}$ è stata utilizzata per modellare il vincolo d uso transitorio
- la variabile $machineMoveCost$ è stata utilizzata per modellare il vincolo di machine move cost
- la variabile $processMoveCost$ è stata utilizzata per modellare il vincolo di process move cost

Variabili	**Definizione**
$x_{p,m} \in \{0,1\}$	Assegnamento finale del processo p nella macchina m
$U_{m,r} \in \mathbb{N}$	Uso della risorsa r nella macchina m
$LC_{m,r} \in \mathbb{N}$	Load cost della risorsa r nella macchina m
$k_{s,l} \in \{0,1\}$	$k_{s,l} = 1$ Assegnamento del servizio s è alla location l
$loadCost_r \in \mathbb{N}$	Load cost della risorsa r in tutte le macchine
$balanceCost_b \in \mathbb{N}$	Balance cost per l'oggetto b
$machineMoveCost \in \mathbb{N}$	Costo delle macchine per lo spostamento dei processi
$processMoveCost \in \mathbb{N}$	Costo dei processi per il loro spostamento
$processServiceMove_s \in \mathbb{N}$	Numero di processi spostati per ogni servizio
$serviceMoveCost \in \mathbb{N}$	Costo per lo spostamento dei servizi
$z_p \in \{0,1\}$	Indica se il processo p è stato spostato dalla macchina iniziale
$j_{p,n} \in \{0,1\}$	$j_{p,n} = 1$ Indica se il processo p si trova nel neightborhood n
$y_{p,m} \in \{0,1\}$	Indica il processo p nell'assegnamento iniziale o finale si trova nella macchina m
$bc_{m,b}$	Calcola il balance cost dell'oggetto b nella macchina m

Tabella 3.3: Definizioni variabili

3.5 Funzione obbiettivo

La funzione obiettivo specifica la grandezza del problema di cui si vuole trovare il valore ottimale. La funzione obiettivo e i vincoli che la soluzione finale deve rispettare sono generalmente espressioni algebriche complesse, costruite a partire dai dati e dalle variabili.

$$\begin{aligned} totalcost &= \sum_{r \in R} weightLoadCost_r * loadCost_r \\ &+ \sum_{b \in B} weightBalanceCost_b * balanceCost_b \\ &+ weightProcessMoveCost * processMoveCost \\ &+ weightServiceMoveCost * serviceMoveCost \\ &+ weightMachineMoveCost * machineMoveCost \end{aligned} \tag{3.1}$$

3.5.1 Load cost

Il vincolo di load cost impone di trovare il massimo tra due valori, tale vincolo peró non è lineare. Per tradurlo in un vincolo lineare si deve scomporlo in due equazioni dove si impone che la variabile sia maggiore uguale di entrambe le quantitá.

Per modellare il load cost è stato necessario introdurre la variabile $lc_{m,r}$. La (3.3) impone che la variabile sia non negativa, la (3.4) calcola il load cost della macchina m per la risorsa r ed infine la (3.2) impone che il load cost per la risorsa r sia la somma di tutti i load cost nelle macchina in cui viene utilizzata la risorsa.

$$loadCost_r = \sum_{m \in M} lc_{m,r} \qquad \forall\, r \in R \tag{3.2}$$

$$lc_{m,r} \geq 0 \qquad \forall\, r \in R, \quad \forall\, m \in M \tag{3.3}$$

$$lc_{m,r} \geq U_{m,r} - SC_{m,r} \qquad \forall\, r \in R, \quad \forall\, m \in M \tag{3.4}$$

3.5.2 Balance cost

Per modellare il *balance cost* è stato necessario introdurre le variabili $bc_{m,b}$. Il vincolo (3.5) impone che queste variabili siano non negative, nel vincolo (3.6) viene calcolato il valore del *balance cost* per la macchina m,infine nel vincolo (3.7) viene calcolato il *balance cost* totale che è rappresentato dalla somma di tutti i *balance cost* su tutte le macchine.

$$bc_{m,b} \geq 0 \qquad \forall\, b \in B \qquad\qquad \forall\, m \in M \tag{3.5}$$

$$bc_{m,b} \leq target(C_{m,r1} - U_{m,r1}) - (C_{m,r2} - U_{m,r2}) \qquad \forall\, b \in B, \forall\, m \in M \tag{3.6}$$

$$balanceCost_b = \sum_{m \in M} bc_{m,b} \qquad\qquad \forall\, b \in B \tag{3.7}$$

3.5.3 Process move cost

Il *proces move cost* è calcolato facendo la somma su tutti i processi e moltiplicando il parametro PMC_p con la variabile z_p che indica se un processo è stato spostato.

$$processMoveCost = \sum_{p \in P} PMC_p z_p \tag{3.8}$$

3.5.4 Service move cost

Il *service move cost* è calcolato dapprima utilizzando l'equazione (3.9) che conta tutti i processi spostati all'interno di un servizio. Infine tramite la (3.10) si sceglie il valore massimo tra tutte le variabile precedentemente calcolate.

$$processServiceMove_s = \sum_{p \in S} z_p \qquad\qquad \forall\, s \in S \tag{3.9}$$

$$serviceMoveCost \geq processServiceMove_s \qquad\qquad \forall\, s \in S \tag{3.10}$$

3.5.5 Machine move cost

Il *machine move cost* viene calcolato facendo una sommatoria per tutti i processi e tutte le macchine del prodotto tra la variabile $x_{p,m}$ e tra il parametro $MMC_{msource,mdestinitation}$ in modo da trovare solo le macchine che effettivamente hanno visto spostarsi dei processi.

$$machineMoveCost = \sum_{p \in P} \sum_{m \neq M0} MMC_{M0,m} x_{p,m} \tag{3.11}$$

3.6 Vincoli

3.6.1 Vincoli di capacitá

Nell'equazione (3.12) viene definita la variabile che indica l'uso della risorsa r nella macchina m. Il vincolo successivo definisce il vincolo di capacitá imponendo che la capacitá utilizzata sia minore della disponibilitá della risorsa nella macchina.

$$U_{m,r} = \sum_{p \in P} R_{p,r} x_{p,m} \qquad \forall r \in R, \forall m \in M \tag{3.12}$$

$$U_{m,r} \leq C_{m,r} \qquad \forall r \in R, \forall m \in M \tag{3.13}$$

3.6.2 Vincoli di conflitto

Il vincolo di conflitto impone che per ogni macchina $m \in M$ e per ogni servizio $s \in S$ sia assegnato al massimo un processo del servizio s alla macchina m.

$$\sum_{p \in S} x_{p,m} \leq 1 \quad \forall s \in S, \quad \forall m \in M \tag{3.14}$$

3.6.3 Vincoli di spread

Nell'equazione (3.15) si impone, per ogni servizio, che ci siano almeno $spreadmin_s$ location in cui processi del servizio s siano stati assegnati. Le equazioni (3.16) e (3.17) impongono invece dei vincoli di dipendenza tra le variabili $x_{p,m}$ e $k_{s,l}$, infatti non puó esserci una variabile $k_{s,l}$ a 1 se nessun processo è stato assegnato a quel servizio e viceversa.

$$\sum_{l \in L} k_{s,l} \geq spreadMin_s \qquad \forall s \in S \tag{3.15}$$

$$k_{s,l} \leq \sum_{p \in S,\, m \in M} x_{p,m} \qquad \forall s \in S \quad \forall l \in L \tag{3.16}$$

$$k_{s,l} \geq x_{p,m} \qquad \forall p \in S \quad \forall m \in l, \quad \forall s \in S \quad \forall l \in L \tag{3.17}$$

3.6.4 Vincoli di dipendenza

L'equazione (3.18) impone il vincolo di dipendenze e richiede che per ogni processo p associato al servizio s^a i processi q associati al servizio s^b si trovano nello stesso neightborhood dei processi nel servizio s^a.

Le equazione (3.19) e (3.19), come per i vincoli di spread, legano le variabili $j_{p,n}$ alle variabili $x_{p,m}$.

$$j_{p,n} \leq \sum_{q \in s^b} j_{q,n} \qquad \forall p \in s^a \quad \forall n \in N, \quad \forall (s^a, s^b) \in B \tag{3.18}$$

$$j_{p,n} \leq \sum_{p \in P} x_{p,m} \qquad \forall m \in M \quad \forall n \in N \tag{3.19}$$

$$j_{p,n} \geq x_{p,m} \qquad \forall p \in P \quad \forall m \in n, \quad \forall n \in N \tag{3.20}$$

3.6.5 Vincoli d'uso transitorio

La prima equazione (3.21) impone, che se un processo p è stato spostato, la richiesta della risorsa r appartenente al sottoinsieme di risorse TR sia minore della capacitá della risorsa nella macchina in cui e' stato spostato. Da notare che, se un processo non viene spostato, questo vincolo si sovrappone al vincolo di capacita.

Le equazioni (3.22) e (3.23) definiscono le variabili $y_{p,m}$ imponendo che siano uguali a 1 se le corrispondenti variabili $x_{p,m}$ sono a 0 se inizialmente il processo p era assegnato alla macchina m.

$$\sum_{p \in P} R_{p,r} y_{p,m} \leq C_{m,r} \qquad \forall r \in TR \quad \forall m \in M \tag{3.21}$$

$$y_{p,m} \geq x_{p,m} \qquad \forall p \in P \quad \forall m \in M \tag{3.22}$$

$$\sum_{m \in M} y_{p,m} M0 B_{p,m} = 1 \qquad \forall p \in P \tag{3.23}$$

3.6.6 Altri vincoli

L'equazione (3.24) definisce la variabile z_p, questa variabile vale 1 se e solo se inizialmente il processo p si trovava in una macchina diversa, infatti $w_{p,m} = 1$ se e solo se il processo p non si trova inizialmente nella macchina m.

L'equazione (3.25) stabilisce che un processo puó trovarsi in una sola macchina.

$$z_p = \sum_{m \in M} w_{p,m} x_{p,m} \qquad \forall p \in P \tag{3.24}$$

$$\sum_{m \in M} x_{p,m} = 1 \qquad \forall p \in P \tag{3.25}$$

Capitolo 4

Analisi dei risultati

4.1 Risultati

La macchina utilizzata per la risoluzione del modello presenta queste caratteristiche:

- Processore Intel core i5 750 2.66 GHz
- 8 GB Ram
- CPLEX v 12.2

Sono state provate 10 istanze numeriche diverse, nella Tabella 4.1 sono riportate, per ogni istanza, il numero di macchine, processi, risorse, servizi, location e neightborhhood.

Nella Tabella 4.2 sono rappresentati il numero di vincoli e variabili generati per ogni istanza.

Istanza	Macchine	Processi	Risorse	Servizi	Location	Neightborhhood
A1_1	4	100	2	79	4	1
A2_2	100	1 000	4	980	4	2
A2_3	100	1 000	3	216	25	5
A2_4	50	1 000	3	142	50	50
A2_5	12	1 000	4	981	4	2
A2_1	100	1 000	3	1 000	1	1
A2_2	100	1 000	12	170	25	5
A2_3	100	1 000	12	129	25	5
A2_4	50	1 000	12	180	25	5
A2_5	50	1 000	12	153	25	5

Tabella 4.1: Istanze

Istanza	Vincoli	Variabili totali	Variabili binarie
A1_1	2 417	1 421	1 316
A1_2	409 896	208 507	206 720
A1_3	346 458	212 222	211 400
A1_4	241 981	167 146	157 200
A1_5	58 070	32 021	30 924
A2_1	408 305	204 606	203 000
A2_2	333 074	212 835	210 250
A2_3	347 746	211 769	209 225
A2_4	182 570	111 916	110 500
A2_5	186 683	111 193	109 825

Tabella 4.2: Numero di vincoli e variabili generati

Delle dieci istanze provate, solo su tre sono state trovate le soluzioni ottime in tempi ragionevoli. Nella Tabella 4.3 sono riportate le tre soluzioni ottime trovate dal risolutore.

Per le istanze in cui non è stato possibile trovare una soluzione ottima viene riportato nella Tabella 4.4

- *Tempo* rappresenta da quanto è in esecuzione il programma
- *Nodi* indica il numero di nodi esaminati
- *Soluzioni ammissibili* indica il numero di soluzioni ammissibili trovate
- *Dimensione problema* visualizza la memoria utilizzata dall'algoritmo branch-and-cut.

Dai precedenti risultati si nota che CPLEX ha difficoltá nel trovare non solo la soluzione ottima ma anche delle soluzioni ammissibili. Si è provato a eseguire le istanze passando come incumbent il valore della funzione obbiettivo calcolata considerando l'assegnamento iniziale. Con l'inizializzazione innanzitutto è stato possibile trovare una soluzione ammissibile per tutte le istanze, i tempi di esecuzione per trovare la soluzione ottima sono diminuiti, inoltre, per alcune istanze, è stato possibile determinare una soluzione migliore rispetto a quella determinata senza inizializzazione. Per le istanze A1_1, A1_3 e A1_5 nel secondo tipo di esecuzione il tempo necessario per determinare la soluzione migliore diminuisce. Per le istanze A2_2, A2_3, A2_4, A2_5 invece è possibile determinare una soluzione ammissibile. Infine per le istanze A1_2, A1_4 e A2_1, a paritá di tempo di esecuzione si riescono a determinare soluzioni migliori rispetto al caso precedente. Nella Tabella 4.5 vengono riportati i risultati ottenuti.

Dai risultati ottenuti si deduce che CPLEX non riesce sempre a trovare una soluzione ammissibile anche se la l'assegnamento di partenza è giá una soluzione ammissibile. Il motivo principale potrebbe essere dovuto non tanto dalle dimensioni del problema ma piú probabilmente dalla struttura del problema non adeguata alle euristiche di default utilizzate dal risolutore.

Per migliorare la risoluzione del modello potrebbe essere opportuno progettare delle euristiche ad-hoc che sfruttano la struttura del problema.

Istanza	Prima soluzione ammissibile trovata		Soluzione ottima		Nodi totali visitati
	Nodo	Valore	Tempo	Valore	
A1_1	0	$3.44159\ 10^7$	0.1s	$4.430\,50\ 10^7$	2
A2_3	0	$5.83662\ 10^8$	25.1 s	$5.830\,08\ 10^8$	68
A2_5	1174	$7.275783\ 10^8$	5.1 s	$7.2754\,78\ 10^8$	1174

Tabella 4.3: Tempo impiegato per determinare la soluzione ottima

Istanza	Tempo	Nodi	Soluzioni ammissibili	Dimensione problema
A1_2	1 478.7 s	13 040	144	812.42 MB
A1_4	2 036.7 s	58 7	23	91.89 MB
A2_1	1 353.9 s	472	6	81.51 MB
A2_2	1 896.1 s	33	0	0.09 MB
A2_3	3 364.1 s	53	0	7.30 MB
A2_4	3 600.0 s	0	0	0.00 MB
A2_5	1 670.5 s	53	0	2.85 MB

Tabella 4.4: Tempi di esecuzione

Istanza	Senza inizializzazione			Con inizializzazione		
	Tempo	**Soluzione**	**Gap**	**Tempo**	**Soluzione**	**Gap**
A1_1	0.1 s	$4.43\ 10^7$	0.0%	0.0 s	$4.43\ 10^7$	0.0%
A1_2	3 600.0 s	$7.95\ 10^8$	2.2%	3 600.0 s	$7.90\ 10^8$	1.6%
A1_3	25.1 s	$5.83\ 10^8$	0.0%	20.4 s	$5.83\ 10^8$	0.0%
A1_4	3 600.0 s	$2.88\ 10^8$	15.8%	3 600.0 s	$2.77\ 10^8$	12.5%
A1_5	5.4 s	$7.28\ 10^8$	0.0%	4.8 s	$7.28\ 10^8$	0.0%
A2_1	3 600.0 s	$9.13\ 10^7$	100.0%	3 600.0 s	$1.14\ 10^7$	100.0%
A2_2	3 600.0 s	–	100.0%	3 600.0 s	$1.16\ 10^9$	54.6%
A2_3	3 600.0 s	–	100.0%	3 600.0 s	$2.27\ 10^9$	54.5%
A2_4	3 600.0 s	–	100.0%	3 600.0 s	$3.22\ 10^9$	47.9%
A2_5	3 600.0 s	–	100.0%	3 600.0 s	$7.48\ 10^8$	59.0%

Tabella 4.5: Soluzioni trovate senza e con inizializzazione

4.2 Esempio

In questa sezione è riportato un esempio di output del risolutore CPLEX.

La prima azione che compie il risolutore è la fase di *presolve* in cui si cerca di ridurre le dimensioni del problema (ad esempio eliminando righe vuote) e di dimostrare che non esiste una soluzione ammissibile senza risolvere il modello.

```
IBM ILOG License Manager: "IBM ILOG Optimization Suite for Academic Initiative"
is accessing CPLEX 12 with option(s): "e m b q ".

<<< generate

Tried aggregator 2 times.
MIP Presolve eliminated 1673 rows and 501 columns.
MIP Presolve modified 40 coefficients.
Aggregator did 360 substitutions.
Reduced MIP has 385 rows, 561 columns, and 2167 nonzeros.
Reduced MIP has 540 binaries, 21 generals, 0 SOSs, and 0 indicators.
Probing time =    0.00 sec.
Tried aggregator 1 time.
MIP Presolve eliminated 40 rows and 0 columns.
Reduced MIP has 345 rows, 561 columns, and 2083 nonzeros.
Reduced MIP has 540 binaries, 21 generals, 0 SOSs, and 0 indicators.
Presolve time =    0.01 sec.
Found feasible solution after 0.02 sec.  Objective = 3.4416e+08
Probing time =    0.00 sec.
Clique table members: 241.
MIP emphasis: balance optimality and feasibility.
MIP search method: dynamic search.
Parallel mode: deterministic, using up to 4 threads.
Root relaxation solution time =    0.06 sec.
```

Figura 4.1:

Nella Figura 4.2 sono riportati tutti i nodi dell'albero visitati dal risolutore. Nella prima colonna viene riportato un asterisco ogni volta che viene trovato una soluzione ammissibile. Le due colonne successive indicano il nodo visitato e il numero di nodo visitato nel sotto albero sinistro. La colonna *Objective* riporta il valore del rilassamento. Un nodo viene scartato se la soluzione del sotto problema è impossibile o se il valore del rilassamento è peggiore dell'upper bound globale. La colonna *IInt* riporta il numero delle variabili che hanno violato il vincolo di interezza. La colonna *Best Integer* riporta il valore della migliore soluzione trovata. a colonna *Cuts Best Node* riporta il valore del migliore rilassamento. Se compare la parola *Cuts* significa che sono stati generati dei piani di taglio. Nella colonna *Iteration count* è riporta il valore delle iterazioni eseguite. Infine viene riportato il *gap*.

```
        Nodes                                         Cuts/
   Node  Left     Objective  IInf  Best Integer    Best Node    ItCnt     Gap

*     0+    0                        3.44159e+08                  223      ---
*     0+    0                        3.44157e+08                  223      ---
*     0+    0                        3.44156e+08                  223      ---
*     0+    0                        3.44154e+08                  223      ---
      0     0   4.43065e+07    16    3.44154e+08   4.43065e+07    223    87.13%
      0     0   4.43065e+07    20    3.44154e+08       Cuts: 5    233    87.13%
      0     0   4.43065e+07    23    3.44154e+08    MIRcuts: 1    235    87.13%
*     0+    0                        4.95288e+07   4.43065e+07    235    10.54%
*     0+    0                        4.56756e+07   4.43065e+07    235     3.00%
      0     2   4.43065e+07    23    4.56756e+07   4.43065e+07    235     3.00%
Elapsed real time =    0.17 sec. (tree size =  0.01 MB, solutions = 6)
*     2     0      integral     0    4.43065e+07   4.43065e+07    263     0.00%
```

Figura 4.2:

Nella Figura 4.3 sono elencate le famiglie di tagli utilizzate da CPLEX per risolvere il modello.

```
GUB cover cuts applied:  1
Mixed integer rounding cuts applied:  2
Gomory fractional cuts applied:  1

Root node processing (before b&c):
  Real time             =    0.15
Parallel b&c, 4 threads:
  Real time             =    0.01
  Sync time (average)   =    0.00
  Wait time (average)   =    0.00
                          -------
Total (root+branch&cut) =    0.16 sec.
```

Figura 4.3:

Infine viene visualizzata il valore della soluzione ottima trovata e l'assegnamento finale dei processi alle macchine.

```
OBJECTIVE: 4.43065e+7
0 0 0 1 1 3 3 3 3 2 1 0 0 3 3 1 3 0 0 3 2 1 1 1 0 1 1 2 0 3 3 3 0 2 2 0 0 1 0 3
0 0 1 3 1 3 0 2 2 2 0 3 0 3 3 0 1 1 2 0 0 2 2 1 0 2 3 1 1 0 0 2 0 0 3 0 1 0 1 3
3 1 0 0 0 3 0 0 0 2 0 3 3 1 2 3 2 1 1 2
<<< post process
```

Figura 4.4:

Appendice A

Sorgenti

A.1 Parametri iniziali

Inizialmente sono stati forniti due file di testo contenenti l'assegnamento iniziale e i parametri necessari per la risoluzione del modello. Di seguito sono riportati i valori di una istanza giocattolo utilizzata per verificare la correttezza del modello.

Assegnamento iniziale

```
0 0 0 1 1 3 3 3 3 2 1 0 0 3 3 1 3 0 0 3 2 1 1 1 0 1 1 2
0 3 3 3 0 2 2 0 0 1 0 3 0 0 1 3 1 3 0 2 2 2 0 3 0 3 3 0
1 1 2 0 0 2 2 1 0 2 3 1 1 0 0 2 0 0 2 0 1 0 1 3 3 1 0 0
0 3 0 0 0 2 0 3 3 1 2 3 2 1 1 2
```

Parametri

```
2
1 100
0  10
4
0 0 30 400 16  80 0 1 4 5
0 0 10 240  8 160 1 0 3 4
```

```
1 1 15 100 12  80 4 3 0 2
1 2 10 100  8  80 5 4 2 0
2
2 0
1 1 0
3
0 12  10 1000
0 10  20  100
1  6 200    1
1
0 1 20
10
1 10 100
```

A.2 Modello OPL

Di seguito è riportato il modello algebrico tradotto in linguaggio OPL.

```
int NbMachine = ...;
int NbProcess = ...;
int NbResource = ...;
int NbService = ...;
int NbLocation = ...;
int NbD=...;
int NbNeighborhodoos = ...;
{int} TR = ...;
int NbB=...;

range Machine=1..NbMachine;
range Process=1..NbProcess;
range Resource=1..NbResource;
range Service=1..NbService;
range Location=1..NbLocation;
range Neighborhodoos=1..NbNeighborhodoos;
range B=1..NbB;
range D=1..NbD;

{int} s[Service] = ...;
{int} l[Location] = ...;
{int} n[Neighborhodoos] = ...;
tuple tuplad{
   int sa;
   int sb;
};
tuplad d[D]=...;
tuple tuplab{
   int r1;
   int r2;
   int target;
}
tuplab b[B]=...;
int M0[Process] =...;
int M0B[Process][Machine]=...;
int C[Machine][Resource]=...;
```

```
int R[Process][Resource]=...;
int spreadMin[Service]=...;
int SC[Machine][Resource]=...;
int PMC[Process]=...;
int MMC[Machine][Machine]=...;
int weightLoadCost[Resource]=...;
int weightBalanceCost[B]=...;
int weightProcessMoveCost=...;
int weightServiceMoveCost=...;
int weightMachineMoveCost=...;
int w[Process][Machine]=...;

dvar boolean xb[Process][Machine];
dvar int+ U[Machine][Resource];
dvar int lc[Machine][Resource];
dvar boolean  k[Service][Location];
dvar int loadCost[Resource];
dvar int balanceCost[B];
dvar int machineMoveCost;
dvar int processMoveCost;
dvar int processMoveService[Service];
dvar int serviceMoveCost;
dvar boolean z[Process];
dvar boolean j[Process][Neighborhodoos];
dvar boolean y[Process][Machine];
dvar int temp[Machine][B];

minimize
   sum ( r in Resource)
      weightLoadCost[r] * loadCost[r]
   + sum (b in B)
        weightBalanceCost[b] * balanceCost[b]
   + weightProcessMoveCost * processMoveCost
   + weightServiceMoveCost * serviceMoveCost
   + weightMachineMoveCost * machineMoveCost;

subject to{

   forall( r in Resource)
```

```
      sum ( m in Machine)
         lc[m][r] == loadCost[r];

forall( r in Resource, m in Machine)
   lc[m][r]>=0;

forall( r in Resource, m in Machine)
   lc[m][r]>=U[m][r]-SC[m][r];

forall( m in Machine, tt in B)
   bc[m][tt] >= 0;

forall (m in Machine, t in B)
   b[t].target * (C[m][b[t].r1] -U[m][b[t].r1])
   - (C[m][b[t].r2] - U[m][b[t].r2]) <= bc[m][t];

forall(b in B)
   sum(m in Machine)
      bc[m][b]==balanceCost[b];

sum(p in Process)
   PMC[p]*z[p]==processMoveCost;

forall( t in Service)
  sum(p in s[t])
    z[p] == processMoveService[t];

forall(s in Service)
  processMoveService[s] <= serviceMoveCost;


sum(p in Process)
  sum (m in Machine)
     MMC[M0[p]][m] * xb[p][m] == machineMoveCost;

forall ( m in Machine, r in Resource)
  sum(p in Process)
    R[p][r] *  xb[p][m] == U[m][r];
```

```
forall( r in Resource, m in Machine)
   U[m][r]<= C[m][r];

forall ( t in Service, m in Machine)
   sum(p in s[t])
      xb[p][m]<=1;

forall ( s in Service)
   sum (l in Location)
     k[s][l] >= spreadMin[s];

forall(t in Service, tt in Location)
   sum (p in s[t],m in l[tt])
      xb[p][m] >= k[t][tt];

forall(t in Service,tt in Location,m in l[tt],p in s[t])
   k[t][tt]>= xb[p][m];

forall(t in D,p in s[d[t].sa], n in Neighborhodoos)
   sum(q in s[d[t].sb] )
      j[q][n]>=j[p][n];

forall( t in Neighborhodoos,m in n[t])
   sum (p in Process)
      xb[p][m]<=
      sum (p in Process) j[p][t];

forall(p in Process, t in Neighborhodoos, m in n[t])
   xb[p][m]<=j[p][t];

forall(r in TR, m in Machine)
   sum(p in Process)
      R[p][r]*y[p][m]<= C[m][r];

forall(p in Process, m in Machine)
  y[p][m]>= xb[p][m];

forall(p in Process)
  sum (m in Machine)
```

```
      y[p][m] * M0B[p][m]==1;

forall (p in Process)
   sum ( m in Machine)
      w[p][m]*xb[p][m]==z[p];

forall( p in Process )
   sum (m in Machine)
      xb[p][m] == 1;

}
{int} x[p in Process]={m | m in Machine : xb[p][m]==1};
execute DISPLAY {
   for(var p in Process) write(Opl.first(x[p])-1," ");
}
```

Dati

File di dati ricavato dai file di testo forniti.

```
NbMachine = 4;
NbProcess = 3;
NbResource = 2;
NbService = 2;
NbLocation = 3;
NbD = 1;
NbNeighborhodoos = 2;
TR = {1};
NbB = 1;
s = [{1,2},
     {3}];
l = [{1,2},
     {3},
     {4}];
n = [{1,2},
     {3,4}];
d = [<2,1>];
b = [<1,2,20>];
M0 = [1,4,1];
M0B = [[1,0,0,0],
       [0,0,0,1],
       [1,0,0,0]];
C = [[30,400],
     [10,240],
     [15,100],
     [10,100]];
R = [[12,10],
    [10,20],
    [6,200]];
spreadMin = [ 2, 1];
SC = [[16,80],
     [8,160],
     [12,80],
     [8,80]];
PMC = [ 1000, 100, 1];
MMC =[[0,1,4,5],
```

```
       [1,0,3,4],
       [4,3,0,2],
       [5,4,2,0]];
weightLoadCost = [100,10];
weightBalanceCost = [10];
weightProcessMoveCost = 1;
weightServiceMoveCost = 10;
weightMachineMoveCost = 100;
w=[[0,1,1,1],
   [1,1,1,0],
   [0,1,1,1]];
```

A.3 Conversione dati

Di seguito sono riportate le tre classi Java utilizzate per convertire i file di dati iniziali in un file di estensione *.dat*.

Elaborazione.java

```
import java.io.*;
import java.io.Reader;
import java.util.*;

public class Elaborazione {
   public static void main(String[] args){
      String in1=args[0];
      String in2=args[1];
      String out="output.dat";
      Formattazione b=new Formattazione(in1, in2, out);
      b.start();
      b.scrivi();
   }
}
```

Estrai.java

```
import java.util.*;
import java.io.*;

public class Estrai {

   private int numeroRisorse;
   private int numeroMacchine;
   private int numeroServizi;
   private int numeroProcessi;
   private int numeroOggettiB;
   private int numeroDipendenze;
   private String dipendenze[];
   private int weightProcessMoveCost;
   private int weightServiceMoveCost;
```

```
    private  int weightMachineMoveCost;
    private String riga[];
    private String riga2[];
    private int TR[];
    private String weightLoadCost[];
    private int numeroRisorseTR;
    private int numeroNeighborhood;
    private String n[];
    private int lunghezzaRigaMacchina = 1000000;
    private int numeroLocation;
    private String l[];
    private String Cmr[];
    private String SCmr[];
    private String movingCost[];
    private String spreadMin;
    private int lunghezzaRigaServizio = 1000000;
    private int lunghezzaRigaProcesso = 1000000;
    private int lunghezzaRigaB = 1000000;
    private String s[];
    private String R[];
    private String PMC;
    private String b[];
    private String weightBalanceCost;
    private String nome;
    private String nome2;
    private String M0p;
    private String M0Bp[];
    private int contatore2;
    private int x[];
    private String w[];

    public Estrai(String nome, String nome2) {
        this.nome = nome;
        this.nome2 = nome2;
    }

    public void informazioni() {
        int contatore = 0;
        try {
```

```
      ArrayList stringList = new ArrayList();
      BufferedReader reader = new BufferedReader(
         new FileReader(new File(nome)));
         String linea = reader.readLine();
         while (linea != null) {
            stringList.add(linea);
            linea = reader.readLine();
            contatore++;
         }
    } catch (Exception e) {
      System.out.println("errore 1");
    }
    riga = new String[contatore];
    for (int i = 0; i < contatore; i++) {
      riga[i] = "";
    }
    try {
      BufferedReader reader = new BufferedReader(
         new FileReader(new File(nome)));
         String linea = reader.readLine();
         int i = 0;
         while (linea != null) {
         riga[i] = linea;
         linea = reader.readLine();
            i++;
         }
    } catch (Exception e) {
         System.out.println("errore");
    }

  for (int i = 0; i < contatore; i++)
    numeroRisorse=Integer.parseInt(riga[0]);
    numeroMacchine=Integer.parseInt(riga[1
        +numeroRisorse]);
    numeroServizi=Integer.parseInt(riga[2
        +numeroRisorse+numeroMacchine]);
    numeroProcessi=Integer.parseInt(riga[3+numeroRisorse
        +numeroMacchine+numeroServizi]);
    numeroOggettiB=Integer.parseInt(riga[4+numeroRisorse
```

```
        +numeroMacchine+numeroServizi+numeroProcessi]);
    String weights = (riga[5 + numeroRisorse
        +numeroMacchine+numeroServizi+numeroProcessi
        +numeroOggettiB*2]);
    String suddividi[]=new String[3];
    suddividi = estraiNumeriDaStringa(weights);
    weightProcessMoveCost=Integer.parseInt(suddividi[0]);
    weightServiceMoveCost=Integer.parseInt(suddividi[1]);
    weightMachineMoveCost=Integer.parseInt(suddividi[2]);

}

public static String[] estraiNumeriDaStringa(String s) {
  int i;
  StringTokenizer st = new StringTokenizer(s, " ");
  String[] stringheSplittate=new String[st.countTokens()];
  i = 0;
  while (st.hasMoreTokens()) {
    stringheSplittate[i] = st.nextToken();
    i++;
  }
  return stringheSplittate;
}

public void elaboraraRisorse() {
   String[] temp = new String[2];
   for (int i = 1; i <= (numeroRisorse + 1); i++) {
      temp = estraiNumeriDaStringa(riga[i]);
      if (temp[0].equals("1"))
         numeroRisorseTR++;
   }
   TR = new int[numeroRisorseTR];
   int k = 0;
   for (int i = 1; i <= (numeroRisorse + 1); i++) {
       temp = estraiNumeriDaStringa(riga[i]);
       if (temp[0].equals("1"))
       {
          TR[k] = i;
          k++;
```

```
        }
    }
    weightLoadCost = new String[numeroRisorse];
    int j = 0;
    for (int i = 1; i < (1 + numeroRisorse); i++) {
       temp = estraiNumeriDaStringa(riga[i]);
       weightLoadCost[j] = "," + (temp[1]);
       j++;
    }
}

public void elaboraNeighborhood() {
    numeroNeighborhood = calcolaNumeroNeighborhood();
    n = new String[numeroNeighborhood];
    for (int i = 0; i < numeroNeighborhood; i++) {
       n[i] = "";
       String temp[] = new String[lunghezzaRigaMacchina];
       for (int i = (2 + numeroRisorse); i < (2
            + numeroRisorse + numeroMacchine); i++) {
         temp = estraiNumeriDaStringa(riga[i]);
         int j = i - 1 - numeroRisorse;
         n[Integer.parseInt(temp[0])] =
                 n[Integer.parseInt(temp[0])]+ "," + j;
      }
   }
}

public void elaboraLocation() {
    numeroLocation = calcolaNumeroLocation();
    l = new String[numeroLocation];
    for (int i = 0; i < numeroLocation; i++)
      l[i] = "";
    String temp[] = new String[lunghezzaRigaMacchina];
    for (int i = (2 + numeroRisorse); i < (2
            + numeroRisorse + numeroMacchine); i++) {
      temp = estraiNumeriDaStringa(riga[i]);
      int j = i - 1 - numeroRisorse;
      l[Integer.parseInt(temp[1])] =
           l[Integer.parseInt(temp[1])] + "," + j;
```

```
    }
}

public void elaboraCmr() {
    Cmr = new String[numeroMacchine];
    for (int i = 0; i < numeroMacchine; i++)
         Cmr[i] = "";
    String temp[] = new String[lunghezzaRigaMacchina];
    for (int i = (2 + numeroRisorse); i < (2
        + numeroRisorse + numeroMacchine); i++) {
      int k = i - 2 - numeroRisorse;
      temp = estraiNumeriDaStringa(riga[i]);
      for (int j = 2; j < (2 + numeroRisorse); j++)
          Cmr[k] = Cmr[k] + "," + temp[j];
    }
}

public void elaboraSCmr() {
   SCmr = new String[numeroMacchine];
   for (int i = 0; i < numeroMacchine; i++)
        SCmr[i] = "";
   String temp[] = new String[lunghezzaRigaMacchina];
   for (int i = (2 + numeroRisorse); i < (2
          + numeroRisorse + numeroMacchine); i++) {
       int k = i - 2 - numeroRisorse;
       temp = estraiNumeriDaStringa(riga[i]);
       for (int j = (2 + numeroRisorse); j < (2
                + numeroRisorse + numeroRisorse); j++)
       SCmr[k] = SCmr[k] + "," + temp[j];
    }
}

public void elaboraMovingCost() {
   movingCost = new String[numeroMacchine];
   for (int i = 0; i < numeroMacchine; i++)
       movingCost[i] = "";
   String temp[] = new String[lunghezzaRigaMacchina];
   for (int i = (2 + numeroRisorse); i < (2
          + numeroRisorse + numeroMacchine); i++) {
```

```
        int k = i - 2 - numeroRisorse;
        temp = estraiNumeriDaStringa(riga[i]);
        for (int j = (2 + numeroRisorse * 2); j < (2 +
              numeroRisorse * 2 + numeroMacchine); j++)
          movingCost[k] = movingCost[k] + "," + temp[j];
    }
}

public void elaboraSpreadMin() {
    spreadMin = "";
    String temp[] = new String[lunghezzaRigaMacchina];
    for (int i = (3 + numeroRisorse + numeroMacchine);
              i <= (2 + numeroRisorse + numeroMacchine +
              numeroServizi); i++) {
        temp = estraiNumeriDaStringa(riga[i]);
        spreadMin = spreadMin + ", " + temp[0];
    }
}

public void elaboraDipendenze() {
    int max=0;
    String temp[] = new String[lunghezzaRigaServizio];
    int j = 0;
    for (int i = (3 + numeroRisorse + numeroMacchine);
             i < (3 + numeroRisorse + numeroMacchine +
             numeroServizi); i++) {
      temp = estraiNumeriDaStringa(riga[i]);
      j = i - 3 - numeroRisorse - numeroMacchine;
      if(Integer.parseInt(temp[1])!=0)
         max = max + Integer.parseInt(temp[1]);
  }
  j=0;
  numeroDipendenze=max;
  dipendenze = new String[numeroDipendenze];
  int k=0;
  int f = 0;
  int ff = 0;
  for (int i = (3 + numeroRisorse + numeroMacchine);
           i < (3 + numeroRisorse + numeroMacchine
```

```
            + numeroServizi); i++) {
        temp = estraiNumeriDaStringa(riga[i]);
        j = i - 3 - numeroRisorse - numeroMacchine;
        if(Integer.parseInt(temp[1])!=0){
          for(int a=2;a<(Integer.parseInt(temp[1])+2);a++){
             f = Integer.parseInt(temp[a]) + 1;
             ff = j + 1;
             dipendenze[k]="<"+ff+","+f+">";
             k++  ;
          }
        }
    }
}

public void elaboraProcessiNeiServizi() {
    s = new String[numeroServizi];
    for (int i = 0; i < numeroServizi; i++)
        s[i] = "";
    String temp[] = new String[lunghezzaRigaProcesso];
    int j = 0;
    for (int i = (4 + numeroRisorse + numeroMacchine
            + numeroServizi); i < (4 + numeroRisorse
            + numeroMacchine + numeroServizi
            + numeroProcessi); i++) {
        temp = estraiNumeriDaStringa(riga[i]);
        j = i - 4 - numeroRisorse - numeroMacchine
            - numeroServizi+1;
        s[Integer.parseInt(temp[0])] =
            s[Integer.parseInt(temp[0])]+ "," + j;
     }
}

public void elaboraRisorsa() {
    R = new String[numeroProcessi];
    for (int i = 0; i < numeroProcessi; i++)
        R[i] = "";
    String temp[] = new String[lunghezzaRigaProcesso];
    int j;
    for (int i = (4 + numeroRisorse + numeroMacchine
```

```
          + numeroServizi); i < (4 + numeroRisorse
          + numeroMacchine + numeroServizi
          + numeroProcessi); i++) {
      temp = estraiNumeriDaStringa(riga[i]);
      j = i - 4 - numeroRisorse - numeroMacchine
           - numeroServizi;
      for (int k = 1; k <= numeroRisorse; k++)
          R[j] = R[j] + "," + temp[k];
   }
}

public void elaboraPMP() {
   String temp[] = new String[lunghezzaRigaProcesso];
   PMC = "";
   for (int i = (4 + numeroRisorse + numeroMacchine +
         numeroServizi); i < (4 + numeroRisorse +
         numeroMacchine + numeroServizi + numeroProcessi)
          ; i++) {
      temp = estraiNumeriDaStringa(riga[i]);
      PMC = PMC + ", " + temp[1 + numeroRisorse];
   }
}

public void elaboraBalanceCost() {
   b = new String[numeroOggettiB];
   for (int i = 0; i < numeroOggettiB; i++)
      b[i] = "";
   String temp[] = new String[lunghezzaRigaB];
   int f = 0;
   for (int i = (5 + numeroRisorse + numeroMacchine
       + numeroServizi + numeroProcessi);  i < (5
       + numeroRisorse + numeroMacchine + numeroServizi
       + numeroProcessi + numeroOggettiB * 2); i++) {
      temp = estraiNumeriDaStringa(riga[i]);
      int j = i - 5 - numeroRisorse - numeroMacchine -
             numeroServizi - numeroProcessi;
      for (int k = 0; k < 3; k++) {
        if(k<2){
           f = Integer.parseInt(temp[k]) + 1;
```

```
            b[j] = b[j] + "," + f;
          }
          else
            b[j] = b[j] + "," + temp[k];
        }
        i++;
    }
}

public void elaboraWeightBalanceCost() {
    weightBalanceCost = "";
    String temp[] = new String[lunghezzaRigaB];
    for (int i = (5 + numeroRisorse + numeroMacchine
                + numeroServizi + numeroProcessi); i <
                (5 + numeroRisorse + numeroMacchine
                + numeroServizi + numeroProcessi +
                (numeroOggettiB * 2)); i++) {
      temp = estraiNumeriDaStringa(riga[i]);
      weightBalanceCost=weightBalanceCost + ","+temp[0];
    }
}

public void elaboraM0P() {
    contatore2 = 0;
    M0p = "";
    x = new int[numeroProcessi];
    try {
      ArrayList stringList = new ArrayList();
      BufferedReader reader = new BufferedReader(
         new FileReader(new File(nome2)));
         String linea = reader.readLine();
         while (linea != null) {
           stringList.add(linea);
           linea = reader.readLine();
           contatore2++;
       }
     } catch (Exception e) {
        System.out.println("Errore");
     }
```

```
    riga2 = new String[contatore2];
    for (int i = 0; i < contatore2; i++)
       riga2[i] = "";
    try {
        BufferedReader reader = new BufferedReader
             (newFileReader(new File(nome2)));
        String linea = reader.readLine();
        int i = 0;
        while (linea != null) {
          riga2[i] = linea;
          linea = reader.readLine();
          i++;
        }
     } catch (Exception e) {
        System.out.println("Errore");
     }
     String temp[] = new String[numeroProcessi];
     for (int i = 0; i < numeroProcessi; i++)
        temp[i] = "";
     for (int i = 0; i < contatore2; i++) {
       temp = estraiNumeriDaStringa(riga2[i]);
       for (int j = 0; j < numeroProcessi; j++) {
         int k=Integer.parseInt(temp[j])+1;
         M0p = M0p + "," + k;
         x[j] = Integer.parseInt(temp[j]);
       }
    }
}

public void elaboraM0Bp() {
    M0Bp = new String[numeroProcessi];
    w = new String[numeroProcessi];
    for (int i = 0; i < numeroProcessi; i++) {
      M0Bp[i] = "";
      w[i]="";
    }
    for (int j = 0; j < numeroProcessi; j++) {
      for (int k = 0; k < numeroMacchine; k++) {
         if (x[j] == (k)) {
```

```
            M0Bp[j] = M0Bp[j] + "," + "1";
              w[j]=w[j]+","+"0";
         } else {
            M0Bp[j] = M0Bp[j] + "," + "0";
            w[j]=w[j]+","+"1";
        }
    }
 }
}

private int calcolaNumeroLocation() {
   int max = 0;
   String temp[] = new String[lunghezzaRigaMacchina];
   for (int i = (2 + numeroRisorse); i < (2
        + numeroRisorse + numeroMacchine); i++) {
      temp = estraiNumeriDaStringa(riga[i]);
      if (Integer.parseInt(temp[1]) > max)
           max = Integer.parseInt(temp[1]);
      return (max + 1);
   }
 }

private int calcolaNumeroNeighborhood() {
   int max = 0;
   String temp[] = new String[lunghezzaRigaMacchina];
   for (int i = (2 + numeroRisorse); i < (2
          + numeroRisorse + numeroMacchine); i++) {
      temp = estraiNumeriDaStringa(riga[i]);
      if (Integer.parseInt(temp[0]) > max)
          max = Integer.parseInt(temp[0]);
   }
   return (max + 1);
}

public String[] getCmr() {
   return Cmr;
  }

  public String[] getSCmr() {
```

```
    return SCmr;
}

public String[] getMovingCost() {
    return movingCost;
}

public String[] getS() {
    return s;
}

public String[] getRisorsa() {
    return R;
}

public String[] getb() {
    return b;
}

public int getWeightMachineMoveCost() {
    return weightMachineMoveCost;
}

public int getNumeroRisorseTR(){
    return numeroRisorseTR;
}

public int getNumeroMacchine(){
    return numeroMacchine;
}

public int getNumeroProcessi(){
  return numeroProcessi;
}

public int getNumeroRisorse(){
    return numeroRisorse;
}
```

```
public int getNumeroServizi(){
   return numeroServizi;
}

public int getNumeroLocation(){
   return numeroLocation;
}

public int getNumeroNeighborhood(){
   return numeroNeighborhood;
}

public int getNumeroOggettiB(){
   return numeroOggettiB;
}

public String[] getL(){
   return l;
}

public String[] getW(){
   return w;
}

public String[] getTR(){
   return TR;
 }

public String[] getN(){
   return n;
}

public String[] getB(){
   return b;
}

public int getNumeroDipendenze(){
   return numeroDipendenze;
}
```

```
  public String[] getDipendenze(){
     return dipendenze;
  }

  public String getSpreadMin(){
    return spreadMin;
  }

  public String getPMC(){
     return PMC;
  }

  public  String getWeightBalanceCost(){
     return weightBalanceCost;
  }

  public String getM0p(){
     return M0p;
  }

  public String[] getM0Bp(){
     return M0Bp;
  }

  public int getWeightProcessMoveCost(){
     return weightProcessMoveCost;
  }

  public int getWeightServiceMoveCost(){
     return weightServiceMoveCost;
  }

}
```

Formattazione.java

```
import java.io.PrintWriter;
```

```
class Formattazione {
   String Macchine;
   String Process;
   String Resource;
   String Service;
   String Location;
   String Neighborhodoos ;
   String TR;
   String B;
   String D;
   String s;
   String l;
   String n;
   String b;
   String d;
   String M0;
   String M0B;
   String C;
   String R;
   String spreadMin;
   String SC;
   String PMC;
   String MMC;
   String weightLoadCost;
   String weightBalanceCost ="" ;
   String weightProcessMoveCost="" ;
   String weightServiceMoveCost="";
   String weightMachineMoveCost="";
   String w="";
   String out;
   Estrai a ;
   String input1;
   String input2;

   public Formattazione(String parametri,String
           assegnamento,String output)
   {
      input1=parametri;
```

```
        input2=assegnamento;
        a =new Estrai(input1,input2);
        out=output;
}

public void start(){
    a.informazioni();
    a.elaboraraRisorse();
    a.elaboraNeighborhood();
    a.elaboraLocation();
    a.elaboraCmr();
    a.elaboraSCmr();
    a.elaboraMovingCost();
    a.elaboraSpreadMin();
    a.elaboraDipendenze();
    a.elaboraProcessiNeiServizi();
    a.elaboraRisorsa();
    a.elaboraPMP();
    a.elaboraBalanceCost();
    a.elaboraWeightBalanceCost();
    a.elaboraM0P();
    a.elaboraM0Bp();
}

public void scrivi(){
    formmattaWeightMoveCost();
    formattaInsiemi();
    formattaS();
    formattaL();
    formattaN();
    formattaB();
    formattaC();
    formattaR();
    formattaDipendenze();
    formattaSpreadMin();
    formattaSC();
    formattaPMC();
    formattaMMC();
    formattaWeightBalanceCost();
```

```
    formattaWeightLoadCost();
    formattaM0p();
    formattaM0Bp();
    formattaW();
    stampa();
}

public void formmattaWeightMoveCost(){
    weightProcessMoveCost= "weightProcessMoveCost = "
        +a.getWeightProcessMoveCost()+";";
    weightServiceMoveCost= "weightServiceMoveCost = "
        +a.getWeightServiceMoveCost()+";";
    weightMachineMoveCost= "weightMachineMoveCost = "
        +a.getWeightMachineMoveCost()+";";
}

public void formattaInsiemi(){
    Neighborhodoos =Neighborhodoos  + "];";
    Macchine = "NbMachine = "+a.getNumeroMacchine()+";";
    Process = "NbProcess = "+a.getNumeroProcessi()+";";
    Resource = "NbResource = "+a.getNumeroRisorse()+";";
    Service = "NbService = "+a.getNumeroServizi()+";";
    Location = "NbLocation = "+a.getNumeroLocation()+";";
    Neighborhodoos  = "NbNeighborhodoos = "
        +a.getNumeroNeighborhood()+";";
    B  = "NbB = "+a.getNumeroOggettiB()+";";
    String tempTR[]=new String[a.getNumeroRisorseTR()];
    tempTR=a.getTR();
    for (int i=0;i<a.getNumeroRisorseTR();i++)
       System.out.println(tempTR);
    if(a.getNumeroRisorseTR()>0){
        TR  = "TR ={"+tempTR[0];
        for( int i=1; i< a.getNumeroRisorseTR(); i++)
            TR =TR  + ","+tempTR[i];
        TR =TR  + "};";
    }
    else
       TR="TR={};";
}
```

```
public void formattaS(){
   s="s = [";
   String tempS[]=new String[a.getNumeroServizi()];
   tempS=a.getS();
   for (int i=0;i<a.getNumeroServizi()-1;i++)
       s=s+"{"+ tempS[i].substring(1)+"},"+ '\n';
   s=s+"{"+tempS[a.getNumeroServizi()-1].substring(1)
        + "}";
   s=s+"];";
}

public void formattaW(){
   w="w=[";
   String tempW[]=new String[a.getNumeroProcessi()];
   tempW=a.getW();
   for (int i=0;i<a.getNumeroProcessi()-1;i++)
      w=w+"["+ tempW[i].substring(1)+"],"+ '\n';
   w=w+"["+tempW[a.getNumeroProcessi()-1].substring(1)
         +"]];";
}

public void formattaL(){
  l= "l = [";
  String tempL[]=new String[a.getNumeroLocation()];
  tempL=a.getL();
  for (int i=0;i<a.getNumeroLocation()-1;i++){
    String temp=tempL[i].substring(1);
    l=l+"{"+ temp+"},"+ '\n';
   }
  l=l+"{"+tempL[a.getNumeroLocation()-1].substring(1)
       +"}";
  l=l+"];";
}

public void formattaN(){
   n= "n = [";
   String tempN[]=new String[a.getNumeroNeighborhood()];
   tempN=a.getN();
```

```
    for (int i=0;i<a.getNumeroNeighborhood()-1;i++){
      String temp=tempN[i].substring(1);
      n=n+"{"+ temp+"},"+ '\n';
    }
    n=n+"{"+tempN[a.getNumeroNeighborhood()-1].
        substring(1)+"}];";
}

public void formattaB(){
    b="b= [";
    String tempB[]=a.getB();
    for (int i=0;i<a.getNumeroOggettiB()-1;i++){
       String temp=tempB[i].substring(1);
       b=b+"<"+ temp+">,"+ '\n';
    }
    if(a.getNumeroOggettiB()>0)
       b=b+"<"+tempB[a.getNumeroOggettiB()-1].
         substring(1)+">];";
    else
       b="b=[];";
}

public void formattaC(){
    C="C= [";
    String tempC[]=new String[a.getNumeroMacchine()];
    tempC=a.getCmr();
    for (int i=0;i<a.getNumeroMacchine()-1;i++){
       String temp=tempC[i].substring(1);
       C=C+"["+ temp+"],"+ '\n';
    }
    C=C+"["+tempC[a.getNumeroMacchine()-1].substring(1)
       +"]];";
}

public void formattaR(){
  R="R= [";
  String tempR[]=new String[a.getNumeroMacchine()];
  tempR=a.getRisorsa();
  for (int i=0;i<a.getNumeroProcessi()-1;i++){
```

```
     String temp=tempR[i].substring(1);
     R=R+"["+ temp+"],"+ '\n';
   }
   R=R+"["+tempR[a.getNumeroProcessi()-1].substring(1)
        +"]];";
}

public void formattaDipendenze(){
    d="d=[";
    D="NbD= "+a.getNumeroDipendenze()+";";
    String tempD[]=new String[a.getNumeroDipendenze()];
    tempD=a.getDipendenze();
    if(a.getNumeroDipendenze()>0){
       for (int i=0;i<a.getNumeroDipendenze()-1;i++)
          d=d+ tempD[i]+","+ '\n';
       d=d+"<"+tempD[a.getNumeroDipendenze()-1].
          substring(1)+"];";
    }
    else
     d="d=[];";
}

public void formattaSpreadMin(){
    spreadMin="spreadMin = ["+a.getSpreadMin().substring(1)
       +"];";
}

public void formattaSC(){
    SC="SC= [";
    String tempSC[]=new String[a.getNumeroMacchine()];
    for (int i=0;i<a.getNumeroMacchine()-1;i++){
       String temp=tempSC[i].substring(1);
       SC=SC+"["+ temp+"],"+ '\n';
    }
    SC=SC+"["+tempSC[a.getNumeroMacchine()-1].substring(1)
      +"]];";
}

public void formattaPMC(){
```

```
    PMC = " PMC = ["+a.getPMC().substring(1)+"];";
}

public void formattaMMC(){
    MMC= "MMC =[";
    String tempMovingCost[]=new
          String[a.getNumeroMacchine()];
    tempMovingCost=a.getMovingCost();
    for (int i=0;i<a.getNumeroMacchine()-1;i++)
       MMC=MMC+"["+tempMovingCost[i].substring(1)
          +"],"+ '\n';
    MMC = MMC+"["+tempMovingCost[a.
          getNumeroMacchine()-1].substring(1)+"]];";
}

public void formattaWeightLoadCost(){
    weightLoadCost= "weightLoadCost =[";
    String tempWeightLoadCost[]=new
         String[a.getNumeroRisorse()];
    for (int i=0;i<a.getNumeroRisorse()-1;i++)
       weightLoadCost = weightLoadCost +
           tempWeightLoadCost[i].substring(1)+",";
    weightLoadCost=weightLoadCost+tempWeightLoadCost[a.
        getNumeroRisorse()-1].substring(1)+"];";
}

public void formattaWeightBalanceCost(){
    weightBalanceCost=" weightBalanceCost = [";
    for (int i=0;i < a.getNumeroOggettiB();i++)
        weightBalanceCost = weightBalanceCost
           +a.getWeightBalanceCost().substring(1);
    weightBalanceCost=weightBalanceCost+"];";
}

public void formattaM0p(){
    M0="M0 =["+a.getM0p().substring(1)+"];";
}

public void formattaM0Bp(){
```

```
  M0B="M0B=[";
  String tempM0Bp[]=new String[a.getNumeroProcessi()];
  tempM0Bp=a.getM0Bp();
  for (int i=0;i<a.getNumeroProcessi()-1;i++)
      M0B=M0B+"["+tempM0Bp[i].substring(1)+"],"+'\n';
  M0B=M0B+"["+tempM0Bp[a.getNumeroProcessi()-1].
        substring(1)+"]];";
}

public void stampa(){
   try{
     PrintWriter scrivi= new PrintWriter(out);
     scrivi.println(Macchine);
     scrivi.println(Process);
     scrivi.println(Resource);
     scrivi.println(Service);
     scrivi.println(Location);
     scrivi.println(D);
     scrivi.println(Neighborhodoos);
     scrivi.println(TR);
     scrivi.println(B);
     scrivi.println(s);
     scrivi.println(l);
     scrivi.println(n);
     scrivi.println(d);
     scrivi.println(b);
     scrivi.println(M0);
     scrivi.println(M0B);
     scrivi.println(C);
     scrivi.println(R);
     scrivi.println(spreadMin);
     scrivi.println(SC);
     scrivi.println(PMC);
     scrivi.println(MMC);
     scrivi.println(weightLoadCost);
     scrivi.println(weightBalanceCost );
     scrivi.println(weightProcessMoveCost+'\n'
         +weightServiceMoveCost+'\n'
         +weightMachineMoveCost);
```

```
      scrivi.println(w);
      scrivi.close();
   }catch(Exception e)
   {
     System.out.println("errore nella scrittura del
     file");
   }
  }
}
```

Elenco delle tabelle

Bibliografia

[1] *ILOG CPLEX 12.1 User s manual*. 2009.

[2] Paolo Serafini. *Ottimizzazione*. Zanichelli, 2000.

[3] Frederick S. Hillier, Gerald J. Lieberman. *Ricerca operativa*. McGraw-Hill, 2006.

[4] Paolo Serafini. *Ricerca operativa*. Springer Verlag, 2009.

[5] http://www.ilog.com.